An Approach to Detect and Prevent SQL Injection Attacks in Database Using Web Service

IndraniBalasundaram [1] Dr. E. Ramaraj[2]

[1]Lecturer, Department of Computer Science, Madurai Kamaraj University, Madurai
[2]Director of Computer Centre Alagappa University, Karaikudi.

Abstract
SQL injection is an attack methodology that targets the data residing in a database through the firewall that shields it. The attack takes advantage of poor input validation in code and website administration. SQL Injection Attacks occur when an attacker is able to insert a series of SQL statements in to a 'query' by manipulating user input data in to a web-based application, attacker can take advantages of web application programming security flaws and pass unexpected malicious SQL statements through a web application for execution by the back-end database. This paper proposes a novel specification-based methodology for the prevention of SQL injection Attacks. The two most important advantages of the new approach against existing analogous mechanisms are that, first, it prevents all forms of SQL injection attacks; second, Current technique does not allow the user to access database directly in database server. The innovative technique "Web Service Oriented XPATH Authentication Technique" is to detect and prevent SQL-Injection Attacks in database the deployment of this technique is by generating functions of two filtration models that are Active Guard and Service Detector of application scripts additionally allowing seamless integration with currently-deployed systems.

General Terms
Languages, Security, Verification, Experimentation.

Keywords
Database security, world-wide web, web application security, SQL injection attacks, Runtime Monitoring

1. Introduction

Information is the most important business asset in today's environment and achieving an appropriate level of Information Security. SQL-Injection Attacks (SQLIA's) are one of the topmost threats for web application security. For example financial fraud, theft confidential data, deface website, sabotage, espionage and cyber terrorism. The evaluation process of security tools for detection and prevention of SQLIA's. To implement security guidelines inside or outside the database it is recommended to access the sensitive databases should be monitored. It is a hacking technique in which the attacker adds SQL statements through a web application's input fields or hidden parameters to gain access to resources or make changes to data. The fear of SQL injection attacks has become increasingly frequent and serious. . SQL-Injection Attacks are a class of attacks that many of these systems are highly vulnerable to, and there is no known fool-proof defend against such attacks. Compromise of these web applications represents a serious threat to organizations that have deployed them, and also to users who trust these systems to store confidential data. The Web applications that are vulnerable to SQL-Injection attacks user inputs the attacker's embeds commands and gets executed [4]. The attackers directly access the database underlying an application and leak or alter confidential information and execute malicious code [1][2]. In some cases, attackers even use an SQL Injection vulnerability to take control and corrupt the system that hosts the Web application. The increasing number of web applications falling prey to these attacks is alarmingly high [3] Prevention of SQLIA's is a major challenge. It is difficult to implement and enforce a rigorous defensive coding discipline. Many solutions based on defensive coding address only a subset of the possible attacks. Evaluation of ""Web Service Oriented XPATH Authentication Technique" has no code modification as well as automation of detection and prevention of SQL Injection Attacks. Recent U.S. industry regulations such as the Sarbanes-Oxley Act [5] pertaining to information security, try to enforce strict security compliance by application vendors.

1.1 SAMPLE - APPLICATION

Application that contain SQL Injection vulnerability. The example refers to a fairly simple vulnerability that could be prevented using a straightforward coding fix. This example is simply used for illustrative purposes because it is easy to understand and general enough to illustrate many different types of attacks. The code in the example uses the input parameters LoginID, password to dynamically build an SQL query and submit it to a database.
For example, if a user submits loginID and password as "secret," and "123," the application dynamically builds and submits the query:

Manuscript received January 5, 2011
Manuscript revised January 20, 2011

SELECT * from FROM user_info WHERE loginID='secret' AND pass1=123

If the loginID and password match the corresponding entry in the database, it will be redirect to user_main.aspx page other wise it will be redirect to error.aspx page.

```
1. dim loginId, Password as string
2. loginId = Text1.Text
3. password = Text2.Text
3. cn.open()
4. qry="select * from user_info where LoginID='" &
loginID & "' and pass1=" & password & ""
5. cmd=new sqlcommand(qry,cn)
6. rd=cmd.executereader()
7. if (rd.Read=True) Then
8. Response.redirect("user_main.aspx")
9. else
10. Response.redirect("error.aspx")
11. end if
12. cn.close()
13. cmd.dispose()
```

Figure 1: Example of .NET code implementation.

1.2 Techniques of SQLIA'S

Most of the attacks are not in isolated they are used together or sequentially, depending on the specific goals of the attacker.

a. Tautologies

Tautology-based attack is to inject code in one or more conditional statements so that they always evaluate to true. The most common usages of this technique are to bypass authentication pages and extract data. If the attack is successful when the code either displays all of the returned records or performs some action if at least one record is returned.

Example: In this example attack, an attacker submits " ' or 1=1 - -"

The Query for Login mode is:

SELECT * FROM user_info WHERE loginID='' or 1=1 -- AND pass1=''

The code injected in the conditional (OR 1=1) transforms the entire WHERE clause into a tautology the query evaluates to true for each row in the table and returns all of them. In our example, the returned set evaluates to a not null value, which causes the application to conclude that the user authentication was successful. Therefore, the application would invoke method user_main.aspx and to access the application [6] [7] [8].

b. Union Query

In union-query attacks, Attackers do this by injecting a statement of the form: UNION SELECT <rest of injected query> because the attackers completely control the second/injected query they can use that query to retrieve information from a specified table. The result of this attack is that the database returns a dataset that is the union of the results of the original first query and the results of the injected second query.

Example: An attacker could inject the text "' UNION SELECT pass1 from user_info where LoginID='secret - -" into the login field, which produces the following query:

SELECT pass1 FROM user_info WHERE loginID='' UNION SELECT pass1 from user_info where LoginID='secret' -- AND pass1=''

Assuming that there is no login equal to "", the original first query returns the null set, whereas the second query returns data from the "user_info" table. In this case, the database would return column "pass1" for account "secret". The database takes the results of these two queries, unions them, and returns them to the application. In many applications, the effect of this operation is that the value for "pass1" is displayed along with the account information

c. Stored Procedures

SQL Injection Attacks of this type try to execute stored procedures present in the database. Today, most database vendors ship databases with a standard set of stored procedures that extend the functionality of the database and allow for interaction with the operating system. Therefore, once an attacker determines which backend database is in use, SQLIAs can be crafted to execute stored procedures provided by that specific database, including procedures that interact with the operating system. It is a common misconception that using stored procedures to write Web applications renders them invulnerable to SQLIAs. Developers are often surprised to find that their stored procedures can be just as vulnerable to attacks as their normal applications [18, 24]. Additionally, because stored procedures are often written in special scripting languages, they can contain other types of vulnerabilities, such as buffer overflows, that allow attackers to run arbitrary code on the server or escalate their privileges.

CREATE PROCEDURE DBO.UserValid(@LoginID varchar2, @pass1 varchar2 AS EXEC("SELECT * FROM user_info WHERE loginID='" +@LoginID+ "' and pass1='" +@pass1+ "'");GO

Example: This example demonstrates how a parameterized stored procedure can be exploited via an SQLIA. In the example, we assume that the query string constructed at lines 5, 6 and 7 of our example has been replaced by a call

to the stored procedure defined in Figure 2. The stored procedure returns a true/false value to indicate whether the user's credentials authenticated correctly. To launch an SQLIA, the attacker simply injects " ' ; SHUTDOWN; --" into either the LoginID or pass1 fields. This injection causes the stored procedure to generate the following query:
SELECT * FROM user_info WHERE loginID='secret' AND pass1='; SHUTDOWN; --
At this point, this attack works like a piggy-back attack. The first query is executed normally, and then the second, malicious query is executed, which results in a database shut down. This example shows that stored procedures can be vulnerable to the same range of attacks as traditional application code [6] [11] [12] [10] [13] [14] [15].

d. Extended stored procedures
IIS(Internet Information Services) Reset

There are several extended stored procedures that can cause permanent damage to a system[19].

Extended stored procedure can be executed by using login form with an injected command as the LoginId

LoginId:';execmaster..xp_xxx;--

Password:[Anything]

LoginId:';execmaster..xp_cmdshell'iisreset';--

Password:[Anything]

select password from user_info where LoginId='; exec master..xp_cmdshell 'iisreset'; --' and Password='

This Attack is used to stop the service of the web server of particular Web application.

Stored procedures primarily consist of SQL commands, while XPs can provide entirely new functions via their code. An attacker can take advantage of extended stored procedure by entering a suitable command. This is possible if there is no proper input validation. xp_cmdshell is a built-in extended stored procedure that allows the execution of arbitrary command lines. For example: exec master..xp_cmdshell 'dir' will obtain a directory listing of the current working directory of the SQL Server process. In this example, the attacker may try entering the following input into a search form can be used for the attack. When the query string is parsed and sent to SQL Server, the server will process the following code:

SELECT * FROM user_info WHERE input text ='' exec master.. xp_cmdshell LoginId /DELETE'--'

Here, the first single quote entered by the user closes the string and SQL Server executes the next SQL statements in the batch including a command to delete a LoginId to the user_info table in the database.

e. Alternate Encodings
Alternate encodings do not provide any unique way to attack an application they are simply an enabling technique that allows attackers to evade detection and prevention techniques and exploit vulnerabilities that might not otherwise be exploitable. These evasion techniques are often necessary because a common defensive coding practice is to scan for certain known "bad characters," such as single quotes and comment operators. To evade this defense, attackers have employed alternate methods of encoding their attack strings (e.g., using hexadecimal, ASCII, and Unicode character encoding). Common scanning and detection techniques do not try to evaluate all specially encoded strings, thus allowing these attacks to go undetected. Contributing to the problem is that different layers in an application have different ways of handling alternate encodings. The application may scan for certain types of escape characters that represent alternate encodings in its language domain. Another layer (e.g., the database) may use different escape characters or even completely different ways of encoding. For example, a database could use the expression char(120) to represent an alternately-encoded character "x", but char(120) has no special meaning in the application language's context. An effective code-based defense against alternate encodings is difficult to implement in practice because it requires developers to consider of all of the possible encodings that could affect a given query string as it passes through the different application layers. Therefore, attackers have been very successful in using alternate encodings to conceal their attack strings.

Example: Because every type of attack could be represented using an alternate encoding, here we simply provide an example of how esoteric an alternatively-encoded attack could appear. In this attack, the following text is injected into the login field: "secret'; exec(0x73687574646f776e) - - ". The resulting query generated by the application is:

SELECT * FROM user_info WHERE loginID='secret'; exec(char(0x73687574646f776e)) -- AND pass1=''

This example makes use of the char() function and of ASCII hexadecimal encoding. The char() function takes as a parameter an integer or hexadecimal encoding of a character and returns an instance of that character. The stream of numbers in the second part of the injection is the

ASCII hexadecimal encoding of the string "SHUTDOWN." Therefore, when the query is interpreted by the database, it would result in the execution, by the database, of the SHUTDOWN command.
References: [6]

f. Deny Database service
This attack used in the websites to issue a denial of service by shutting down the SQL Server. A powerful command recognized by SQL Server is SHUTDOWN WITH NOWAIT [19]. This causes the server to shutdown, immediately stopping the Windows service. After this command has been issued, the service must be manually restarted by the administrator.

select password from user_info where LoginId=';shutdown with nowait; --' and Password='0'

The '--' character sequence is the 'single line comment' sequence in Transact - SQL, and the ';' character denotes the end of one query and the beginning of another. If he has used the default sa account, or has acquired the required privileges, SQL server will shut down, and will require a restart in order to function again. This attack is used to stop the database service of a particular web application.

Select * from user_info where LoginId='1;xp_cmdshell 'format c:/q /yes '; drop database mydb; --AND pass1 = 0

This command is used to format the C:\ drive used by the attacker.

2. Related Work

There are existing techniques that can be used to detect and prevent input manipulation vulnerabilities.

2.1 Web Vulnerability Scanning

Web vulnerability scanners crawl and scan for web vulnerabilities by using software agents. These tools perform attacks against web applications, usually in a black-box fashion, and detect vulnerabilities by observing the applications' response to the attacks [18].However, without exact knowledge about the internal structure of applications, a black-box approach might not have enough test cases to reveal existing vulnerabilities and also have false positives.

2.2 Intrusion Detection System (IDS)

Valeur and colleagues [17] propose the use of an Intrusion Detection System (IDS) to detect SQLIA. Their IDS system is based on a machine learning technique that is trained using a set of typical application queries. The technique builds models of the typical queries and then monitors the application at runtime to identify queries that do not match the model in that it builds expected query models and then checks dynamically-generated queries for compliance with the model. Their technique, however, like most techniques based on learning, can generate large number of false positive in the absence of an optimal training set.
Su and Wassermann [8] propose a solution to prevent SQLIAs by analyzing the parse tree of the statement, generating custom validation code, and wrapping the vulnerable statement in the validation code. They conducted a study using five real world web applications and applied their SQLCHECK wrapper to each application. They found that their wrapper stopped all of the SQLIAs in their attack set without generating any false positives. While their wrapper was effective in preventing SQLIAs with modern attack structures, we hope to shift the focus from the structure of the attacks and onto removing the SQLIVs.

2.3 Combined Static and Dynamic Analysis.

AMNESIA is a model-based technique that combines static analysis and runtime monitoring [1][7]. In its static phase, AMNESIA uses static analysis to build models of the different types of queries an application can legally generate at each point of access to the database. In its dynamic phase, AMNESIA intercepts all queries before they are sent to the database and checks each query against the statically built models. Queries that violate the model are identified as SQLIA's and prevented from executing on the database. In their evaluation, the authors have shown that this technique performs well against SQLIA's. The primary limitation of this technique is that its success is dependent on the accuracy of its static analysis for building query models. Certain types of code obfuscation or query development techniques could make this step less precise and result in both false positives and false negatives
Livshits and Lam [16] use static analysis techniques to detect vulnerabilities in software. The basic approach is to use information flow techniques to detect when tainted input has been used to construct an SQL query. These queries are then flagged as SQLIA vulnerabilities. The authors demonstrate the viability of their technique by using this approach to find security vulnerabilities in a benchmark suite. The primary limitation of this approach is that it can detect only known patterns of SQLIA's and,

because it uses a conservative analysis and has limited support for untainting operations, can generate a relatively high amount of false positives.
Wassermann and Su propose an approach that uses static analysis combined with automated reasoning to verify that the SQL queries generated in the application layer cannot contain a tautology [9]. The primary drawback of this technique is that its scope is limited to detecting and preventing tautologies and cannot detect other types of attacks.

3. Proposed Technique

This Technique is used to detect and prevent SQLIA's with runtime monitoring. The solution insights behind the technique are that for each application, when the login page is redirected to our checking page, it was to detect and prevent SQL Injection attacks without stopping legitimate accesses. Moreover, this technique proved to be efficient, imposing only a low overhead on the Web applications. The contribution of this work is as follows: A new automated technique for preventing SQLIA's where no code modification required, Webservice which has the functions of db_2_XMLGenrerator and XPATH_ Validator such that it is an XML query language to select specific parts of an XML document. XPATH is simply the ability to traverse nodes from XML and obtain information. It is used for the temporary storage of sensitive data's from the database, Active Guard model is used to detect and prevent SQL Injection attacks. Service Detector model allow the Authenticated or legitimate user to access the web applications. The SQLIA's are captured by altered logical flow of the application. Innovative technique (figure:1) monitors dynamically generated queries with Active Guard model and Service Detector model at runtime and check them for compliance. If the Data Comparison violates the model then it represents potential SQLIA's and prevented from executing on the database.
This proposed technique consists of two filtration models to prevent SQLIA'S. 1) Active Guard filtration model 2) Service Detector filtration model. The steps are summarized and then describe them in more detail in following sections.
a. Active Guard Filtration Model
Active Guard Filtration Model in application layer build a Susceptibility detector to detect and prevent the Susceptibility characters or Meta characters to prevent the malicious attacks from accessing the data's from database.

b. Service Detector Filtration Model
Service Detector Filtration Model in application layer validates user input from XPATH_Validator where the Sensitive data's are stored from the Database at second level filtration model. The user input fields compare with the data existed in XPATH_Validator if it is identical then the Authenticated /legitimate user is allowed to proceed.

c. Web Service Layer
Web service builds two types of execution process that are DB_2_Xml generator and XPATH_ Validator. DB_2_Xml generator is used to create a separate temporary storage of Xml document from database where the Sensitive data's are stored in XPATH_ Validator, The user input field from the Service Detector compare with the data existed in XPATH_ Validator, if the data's are similar XPATH_ Validator send a flag with the count iterator value = 1 to the Service Detector by signifying the user data is valid.

Procedures Executed in Active Guard

```
Function stripQuotes(ByVal strWords)
  stripQuotes = Replace(strWords, "'", "''")
  Return stripQuotes
  End Function
  Function killChars(ByVal strWords)
     Dim arr1 As New ArrayList
     arr1.Add("select")
     arr1.Add("--")
     arr1.Add("drop")
     arr1.Add(";")
     arr1.Add("insert")
     arr1.Add("delete")
     arr1.Add("xp_")
     arr1.Add("'")
     Dim i As Integer
     For i = 0 To arr1.Count - 1
      strWords = Replace(strWords, arr1.Item(i), "", , , CompareMethod.Text)
     Next
     Return strWords
  End Function
```

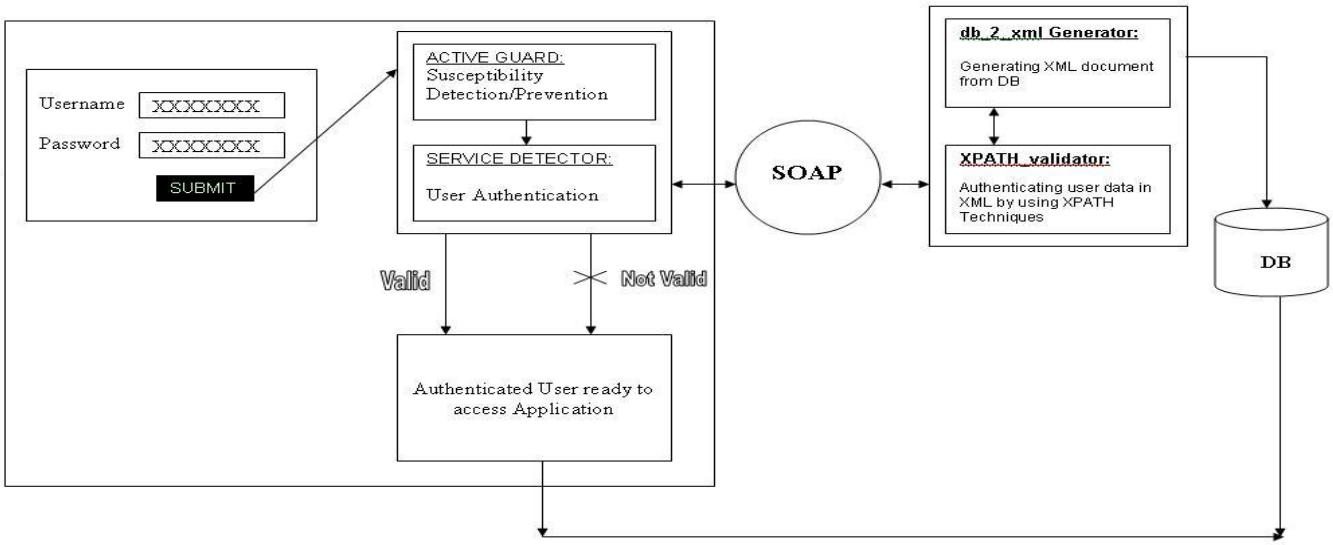

Figure 2: proposed Architecture

Procedures Executed in Service Detector

```
<WebMethod()> _
Public Sub Db_2_XML()
adapt=New SqlDataAdapter("select LoginId,Password from user_info", cn)
dst = New DataSet("Main_Tag")
adapt.Fill(dst, "Details")
dst.WriteXml(Server.MapPath("XML_DATA\XML_DATA.xml"))
End Sub
```

Procedures Executed in Web Service

```
<WebMethod(EnableSession:=True)> _
Public Function XPath_XML_Validation(ByVal userName As String, ByVal Password As Integer) As Integer
Dim xpathdoc As New XPathDocument(Server.MapPath("XML_DATA\XML_DATA.xml"))
Dim navi As XPathNavigator = xpathdoc.CreateNavigator()
Dim expr As XPathExpression =
navi.Compile("/Main_Tag/Details[LoginId='" & userName & "' and Password=" & Password & "]")
Dim nodes As XPathNodeIterator = navi.Select(expr)
Dim count2 As Integer = nodes.Count.ToString()
Return count2
End Function
```

d. Identify hotspot

This step performs a simple scanning of the application code to identify hotspots. Each hotspot will be verified with the Active Server to remove the susceptibility character the sample code (figure: 2) states two hotspots with a single query execution.(In .NET based applications, interactions with the database occur through calls to specific methods in the System.Data.Sqlclient namespace, 1 such as Sqlcommand- . ExecuteReader (String)) the hotspot is instrumented with monitor code, which matches dynamically generated queries against query models. If a generated query is matched with Active Guard, then it is considered an attack.

3.1 Comparison of Data at Runtime Monitoring

When a Web application fails to properly sanitize the parameters, which are passed to, dynamically created SQL statements (even when using parameterization techniques) it is possible for an attacker to alter the construction of back-end SQL statements.

When an attacker is able to modify an SQL statement, the statement will execute with the same rights as the application user; when using the SQL server to execute commands that interact with the operating system, the process will run with the same permissions as the component that executed the command (e.g., database server, application server, or Web server), which is often highly privileged. Current technique (Figure: 1) append with Active Guard, to validate the user input fields to detect the Meta character and prevent the malicious attacker. Transact-SQL statements will be prohibited directly from user input. For each hotspot, statically build a Susceptibility detector in Active Guard to check any malicious strings or characters append SQL tokens (SQL keywords and operators), delimiters, or string tokens to the legitimate command. Concurrently in Web service the DB_2_Xml Generator generates a XML document from database and stored in X_PATH Validator. Service Detector receive the validated user input from Active Guard and send through the protocol SOAP (Simple Object Access Protocol) to the web service from the web service the user input data compare with XML_Validator if it is identical the XML_Validator send a flag as a iterator count value = 1 to Service Detector through the SOAP protocol then the legitimate/valid user is Authenticated to access the web application, If the data mismatches the XML_Validator send a flag as a count value = 0 to Service Detector through the SOAP protocol then the illegitimate/invalid user is not Authenticated to access the web application. In figure 3: In the existing technique query validation occur to validate a Authenticated user and the user directly access the database but in the current technique, there is no query validation .From the Active Guard the validated user input fields compare with the Service Detector where the Sensitive data is stored, db_2_XML Generator is used to generate a XML file and initialize to the class XPATH document the instance Navigator is used to search by using cursor in the selected XML document. With in the XPATH validator, Compile is a method which is used to match the element with the existing document. The navigator will be created in the xpathdocument using select method result will be redirected to the XPATH node iterator. The node iterator count value may be 1 or 0, If the flag value result in Service Detector as 1 then the user consider as Legitimate user and allowed to access the web application as the same the flag value result in Service Detector as 0 then the user consider as Malicious user and reject/discard from accessing the web application If the script builds an SQL query by concatenating hard-coded strings together with a string entered by the user, As long as injected SQL code is syntactically correct, tampering cannot be detected programmatically. String concatenation is the primary point of entry for script injection Therefore, we Compare all user input carefully with Service Detector (Second filtration model). If the user input and Sensitive data's are identical then executes constructed SQL commands in the Application server. Existing techniques directly allows accessing the database in database server after the Query validation. Web Service Oriented XPATH Authentication Technique does not allow directly to access database in database server.

4. EVALUATIONS

The proposed technique is deployed and tried few trial runs on the web server.

Table 1: SQLIA'S Prevention Accuracy

SQL Injection Types	Unprotected	Protected
1. TAUTOLOGIES	Not Prevented	Prevented
2.PIGGY BACKED QUERIES	Not Prevented	Prevented
3. STORED PROCEDURE	Not Prevented	Prevented
4. ALTERNATIVE ENCODING	Not Prevented	Prevented
5. UNION	Not Prevented	Prevented

Table 2: Execution Time comparison for proposed technique

Total Number of Entries in Database	Execution Time in Millisecond	
	Existing Technique	Proposed Technique
1000	1640000	46000
2000	1420000	93000
3000	1040000	46000
4000	1210000	62000
5000	1670000	78000
6000	1390000	107000

The above given table 2 illustrate the execution time taken for the proposed technique with the existing technique.

4.1 SQLIA Prevention Accuracy

Both the protected and unprotected web Applications are tested using different types of SQLIA's; namely use of Tautologies, Union, Piggy-Backed Queries, Inserting additional SQL statements, Second-order SQL injection and various other SQLIA s. Table 1 shows that the proposed technique prevented all types of SQLIA s in all cases. The proposed technique is thus a secure and robust solution to defend against SQLIA's

4.2 Execution Time at Runtime Validation

The runtime validation incurs some overhead in terms of execution time at both the Web Service Oriented XPATH Authentication Technique and SQL-Query based Validation Technique. Taken a sample website E-Transaction measured the extra computation time at the query validation, this delay has been amplified in the graph (figure: 4 and figure:5) to distinguish between the Time delays using bar chart shows that the data validation in XML_Validator performs better than query validation. In Query validation(figure:5) the user input is generated as a query in script engine then it gets parsed in to separate tokens then the user input is compared with the statistical generated data if it is malicious generates error reporting. Web Service Oriented XPATH Authentication Technique (figure: 4) states that user input is generated as a query in script engine then it gets parsed in to separate tokens, and send through the protocol SOAP to Susceptibility Detector, then the validated user data is sequentially send to Service Detector through the protocol SOAP then the user input is compared with the sensitive data, which is temporarily stored in dataset. If it is malicious data, it will be prevented otherwise the legitimate data is allowed to access the Web application.

5. CONCLUSION

SQL Injection Attacks attempts to modify the parameters of a Web-based application in order to alter the SQL statements that are parsed to retrieve data from the database. Any procedure that constructs SQL statements could potentially be vulnerable, as the diverse nature of SQL and the methods available for constructing it provide a wealth of coding options.

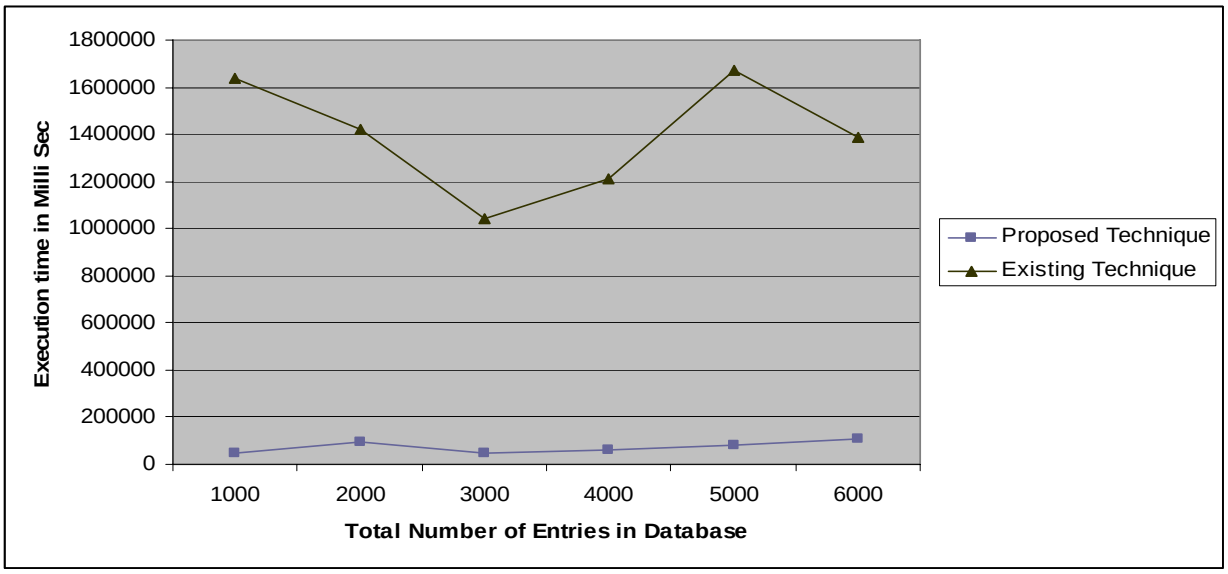

Figure4: Execution Time comparison for proposed technique (data validation in X-path) with existing technique

The primary form of SQL injection consists of direct insertion of code into parameters that are concatenated with SQL commands and executed. This technique is used to detect and prevent the SQLI flaw (Susceptibility characters & exploiting SQL commands) in Susceptibility Detector and prevent the Susceptibility attacker Web Service Oriented XPATH Authentication Technique checks the user input with valid database which is stored separately in XPATH and do not affect database directly then the validated user input field is allowed to access the web application as well as used to improve the performance of the server side validation This proposed technique was able to suitably classify the attacks that performed on the applications without blocking legitimate accesses to the database (i.e., the technique produced neither false positives nor false negatives). These results show that our technique represents a promising approach to countering SQLIA's and motivate further work in this direction

References

[1] William G.J. Halfond and Alessandro Orso , "AMNESIA: Analysis and Monitoring for Neutralizing SQLInjection Attacks", ASE'05, November 7–11, 2005

[2] William G.J. Hal fond and Alessandro Orso, "A Classification of SQL injection attacks and countermeasures",proc IEEE int'l Symp. Secure Software Engg., Mar. 2006.

[3] Muthuprasanna, Ke Wei, Suraj Kothari, "Eliminating SQL Injection Attacks - A TransparentDefenceMechanism", SQL Injection Attacks Prof. Jim Whitehead CMPS 183. Spring 2006, May 17, 2006
[4] William G.J. Hal fond, Alessandro Orso, "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation IEEE Software Engineering, VOL. 34, NO.1January/February 2008
[5] K. Beaver, "Achieving Sarbanes-Oxley compliance for Web applications", http://www.spidynamics.com/support/whitepapers/, 2003
[6] C. Anley, "Advanced SQL Injection In SQL Server Applications," White paper, Next Generation Security Software Ltd., 2002.
[7] W. G. J. Halfond and A. Orso, "Combining Static Analysis and Runtime Monitoring to Counter SQL Injection Attacks," 3rd International Workshop on Dynamic Analysis, 2005, pp. 1- 7
[8] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications," 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2006, pp. 372-382.
[9] G. Wassermann and Z. Su. An Analysis Framework for Security in Web Applications. In Proceedings of the FSE Workshop on Specification and Verification of component-Based Systems (SAVCBS 2004), pages 70–78, 2004.
[10] P. Finnigan, "SQL Injection and Oracle - Parts 1 & 2," Technical Report, Security Focus, November 2002. http://securityfocus.com/infocus/1644
[11] F. Bouma, "Stored Procedures are Bad, O'kay," Technical report, Asp.Net Weblogs, November 2003. http://weblogs.asp.net/fbouma/archive/2003/11/18/38178.aspx.
[12] E. M. Fayo, "Advanced SQL Injection in Oracle Databases," Technical report, Argeniss Information Security, Black Hat Briefings, Black Hat USA, 2005.
[13] C. A. Mackay, "SQL Injection Attacks and Some Tips on How to Prevent them," Technical report, The Code Project, January 2005. http://www.codeproject.com/cs/database/qlInjectionAttacks.asp.
[14] S. McDonald. SQL Injection: Modes of attack, defense, and why it matters. White paper, GovernmentSecurity.org, April 2002. http://www.governmentsecurity.org/articles/SQLInjectionModesofAttackDefenceandWhyItMatters.php
[15] S. Labs. SQL Injection. White paper, SPI Dynamics, Inc.,2002. http://www.spidynamics.com/assets/documents/WhitepaperSQLInjection.pdf.
[16] V. B. Livshits and M. S. Lam. Finding Security Errors in Java Programs with Static Analysis. In Proceedings of the 14th Usenix Security Symposium, pages 271–286, Aug. 2005.
[17] F. Valeur and D. Mutz and G. Vigna "A Learning-Based Approach to the Detection of SQL Attacks," In Proceedings of the Conference on Detection of Intrusions and Malware Vulnerability Assessment (DIMVA), July 2005.
[18] Kals, S., Kirda, E., Kruegel, C., and Jovanovic, N. 2006. SecuBat: a web vulnerability scanner. In Proceedings of the 15th International Conference on World Wide Web. WWW '06. ACM Press, pp. 247-256.
[19] Sql injection - HSC Guides - Web App Security Written by Ethical Hacker sunday, 17 February 2008. http://sqlinjections.blogspot.com/2009/04/sql-injection-hsc-guides-web-app.html.

Prof.E.Ramaraj is presently working as a Technology Advisor, Madurai Kamaraj University, Madurai, Tamilnadu, India on lien from Director, computer centre at Alagappa university, Karaikudi. He has 22 years teaching experience and 8 years research experience. He has presented research papers in more than 50 national and international conferences and published more than 55 papers in national and international journals. His research areas include Data mining, software engineering, database and network security.

B.Indrani received the B.Sc. degree in Computer Science, in 2002; the M.Sc. degree in Computer Science and Information Technology, in 2004. She had completed M.Phil. in Computer Science. She worked as a Research Assistant in Smart and Secure Environment Lab under IIT, Madras. Her current research interests include Database Security.